Projet Mathématique pour informatique

Tristan Moers et Youri Mouton

December 2016

1 Introduction

Dans le cadre du cours de Mathématique pour informatique, il nous est demandé d'implémenter et de tester deux algorithmes résolvant le problème du plus court chemin sur des graphes. Le premier algorithme est celui de Dijkstra et le second est laissé à notre libre choix. L'implémentation se fait en Julia.

2 Utilisation

Les algorithmes peuvent être lancés en Julia comme ceci:

```
1  # Load graph from a file where vertices are indexed from 1 to N
2  g = loadgraph("/Users/youri/Downloads/graph.gml")
3  # Get the shortest path from 1 on the graph using the Dijkstra algorithm
4  d = DijkstraSP(g, 1)
5  # Print the cost to go to 5
6  println(d.dist(5))
7  # Print the path vector from 1 to 5 on the graph
8  println(d.pathTo(5))
9
10
11 # Get the shortest path from 1 on the graph using the Floyd-Warshall algorithm
12 fw = FloydWarshallSP(g)
13 # Print the cost to go to 5
14 println(fw.dist(1, 5))
15 # Print the path vector from 1 to 5 on the graph
16 println(fw.path(1, 5))
```

Néanmoins une fonction cout(i, j) est disponible permettant de lancer les algorithmes de Dijkstra et Floyd-Warshall.

La syntaxe du fichier .gml est décrite au point 3.2.

3 Méthodes

Nous utilisons des types composites dans lesquels sont décrits les constructeurs. Il y a un constructeur pour chaque type contenant des méthodes.

3.1 Edge

Ce type représente une arête rejoignant deux points. Les deux points sont caractérisés par la source et la destination. Le coût de cette liaison est également présent.

3.2 EdgeWeightedGraph

Ce type contient la matrice d'adjacence du graphe représentée ici comme un Array d'Array du type composite Edge, permettant de rendre le code plus clair. Le nombre de noeuds et d'arêtes sont aussi présents.

Une fonction est implémentée afin de récupérer des fichiers exemples contenant un graphe. Ces fichiers sont au format GML. Voici un exemple d'arête:

```
1   edge
2   [
3     source 1
4     target 2
5     value 5
6   ]
```

3.3 Loadgraph

Cette fonction charge en mémoire le fichier .gml contenant le graphe. Elle dénombre les arêtes permettant de construire la matrice d'adjacence du graphe. L'implémentation de cette fonction est disponible à la ligne 270 du code en annexe.

4 Dijkstra

Pour cet algorithme, nous utilisons une PriorityQueue orientée minimum, un tableau des coûts et arêtes pour les noeuds. Les méthodes implémentées sont décrites dans les points suivants.

4.1 relax

Cette méthode parcourt le graphe du point source au noeuds voisins liés par une arête. La distance et le noeud parents sont mis à jour dans le cas ou le coût est moins grand que celui présent dans le tableau de coûts déjà présent.

```
this.relax = function (v::Int)
  # For each vertex neighbour
  for e in g.adj[v]
    w = e.to
    alt = distTo[v] + e.weight
    # If the weight path from source + the edge's weight is smaller than
    # the distance recorded to w, update it
    if (distTo[w] > alt)
      # Update distance to vertex
      distTo[w] = alt
      # Update path to vertex
      vertexTo[w] = e.from
      # Change vertex priority
      pq[w] = distTo[w]
    end
  end
end
```

4.2 dist

Cette fonction retourne le poids total de la destination au noeud source.

```
this.dist = function (to::Int)
  return distTo[to]
end
```

4.3 hasPathTo

Cette fonction permet de vérifier si il y a un chemin vers la destination.

```
this.hasPathTo = function (to::Int)
    return distTo[to] < Inf
end
```

4.4 pathTo

Cette fonction retourne le vecteur contenant les noeuds par lesquels on passe pour aller vers la destination en empruntant le chemin le plus court.

```
this.pathTo = function (to::Int)
    if (!this.hasPathTo(to))
        return
    end
    # Build array of vertex indices going back from destination to source
    path = Int[]
    # Get the vertex index closest to destination
```

```
8       from = vertexTo[to]
9       while (from != 0)
10          # Push the vertex index on the array
11          push!(path, from)
12          # Get the parent vertex index
13          from = vertexTo[from]
14      end
15      # Reverse to obtain natural ordering on the array
16      reverse!(path)
17      # Add the destination vertex index
18      push!(path, to)
19
20      return path
21  end
```

Fonctionnement de l'algorithme: Au départ la PriorityQueue est initialisée ainsi que les deux tableaux. On met à l'infini chaque distance dans le tableau sauf à l'indice de la source où l'on met 0. Et 0 à chaque vertex dans l'autre tableau.
La PriorityQueue est orientée minimum. La prioritée de la source est mise à 0. On parcourt le graphe tant que la PriorityQueue n'est pas vide, en utilisant dequeue à chaque fois. Les tableaux des coûts et arêtes est mis à jour quand un chemin plus court est trouvé.

L'implémentation de cet algorithme est disponible à la ligne 124 du code en annexe.

5 Floyd-Warshall

Pour cet algorithme comme pour le précédent nous utilisons deux tableaux pour les coûts et les arêtes, mais pas de PriorityQueue. Pour les fonctions, celles-ci sont implémentées à la suite :

5.1 path

Cette fonction retourne un vecteur avec les indices des noeuds vers la destination.

```
1   this.path = function (from::Int, to::Int)
2       # Return if there is no path
3       if (!this.hasPath(from, to))
4           return
5       end
6       # Build vector going back from destination to source
7       path = Int[]
8       # Get the parent vector closest to destination
9       parent = vertexTo[from][to]
```

```
10      while (parent != 0)
11          # Add the vertex to the vector
12          push!(path, parent)
13          # Fetch parent vertex
14          parent = vertexTo[from][parent]
15      end
16      # Reverse the vector order to get natural ordering
17      reverse!(path)
18      # Add destination
19      push!(path, to)
20      return path
21  end
```

5.2 dist

Cette fonction retourne le poids total du chemin le plus court entre la source et la destination.

```
1  this.dist = function (from::Int, to::Int)
2      return distTo[from][to]
3  end
```

5.3 hasPath

Cette fonction permet de vérifier si il y a un chemin de la source vers la destination.

```
1  this.hasPath = function (from::Int, to::Int)
2      return distTo[from][to] < Inf
3  end
```

Au départ les deux tableaux sont initialisés. On initialise ensuite tous les chemins à l'infini et les noeuds parents à 0. On insère enfin les coûts et sources dans ces mêmes tableaux.

On parcourt avec trois boucles les noeuds. Dans la troisième boucle, on calcule le coût vers le noeud voisin additionné au coût jusqu'à la source.
Si le coût entre le noeud actuel et son voisin est plus grand que le coût calculé précédemment, la distance entre le noeud et son voisin devient ce coût calculé auparavant. Et le noeud parent est mise à jour.

6 Complexité

6.1 Dijkstra

Si le graphe possède E arcs et V noeuds,que le graphe est représenté par une matrice d'adjacence, alors la complexité temporelle de l'algorithme est ElogV.

6.2 Floyd-Warshall

La complexité temporelle du cet algorithme est V^3. Il est moins performant que Dijkstra qui bénéficie de l'utilisation d'une priority queue avec des accès et ajouts/supressions rapides.

7 Annexe - Julia

```
1  using Base.Collections
2
3  #
4  # Graph Edge
5  #
6  #
7  #
8  type Edge
9    # Variables
10   from::Int       # source vertex
11   to::Int         # destination vertex
12   weight::Float64 # edge weight
13
14   # Constructor
15   function Edge(from::Int, to::Int, weight::Float64)
16     this = new()
17     this.from = from
18     this.to = to
19     this.weight = weight
20
21     return this
22   end
23 end
24
25 #
26 # Edge Weighted undirected graph
27 #
28 #
29 #
30 type EdgeWeightedGraph
```

```
31    # Variables
32    V::Int                  # number of vertices
33    E::Int                  # number of edges
34    adj::Array{Array{Edge}} # adjacency matrix
35
36    # Constructor
37    function EdgeWeightedGraph(V::Int, E::Int, adj::Array{Array{Edge}})
38      this = new()
39      this.V = V
40      this.E = E
41      this.adj = adj
42
43      return this
44    end
45  end
46
47  #
48  # Shortest path Dijkstra algorithm
49  #
50  #
51  #
52  type DijkstraSP
53    # Variables
54    pq::PriorityQueue      # minimum oriented priority queue
55    distTo::Array{Float64} # array of weights to vertices
56    vertexTo::Array{Edge}  # array of edges to vertices
57
58    # Methods
59    relax::Function
60    pathTo::Function
61    dist::Function
62    hasPathTo::Function
63
64    # Constructor
65    function DijkstraSP(g::EdgeWeightedGraph, s::Int)
66      this = new()
67
68      # Core of the algorithm, walk the graph
69      this.relax = function (v::Int)
70        # For each vertex neighbour
71        for e in g.adj[v]
72          w = e.to
73          alt = distTo[v] + e.weight
74          # If the weight path from source + the edge's weight is smaller than
75          # the distance recorded to w, update it
76          if (distTo[w] > alt)
```

```
77            # Update distance to vertex
78            distTo[w] = alt
79            # Update path to vertex
80            vertexTo[w] = e.from
81            # Change vertex priority
82            pq[w] = distTo[w]
83          end
84        end
85      end
86
87      # Returns true if there is a path to destination
88      this.hasPathTo = function (to::Int)
89        return distTo[to] < Inf
90      end
91
92      # Path to a vertex
93      # returns a vector of nodes to the destination
94      this.pathTo = function (to::Int)
95        # Return if there is no path
96        if (!this.hasPathTo(to))
97          return
98        end
99
100       # Build array of vertex indices going back from destination to source
101       path = Int[]
102       # Get the vertex index closest to destination
103       from = vertexTo[to]
104       while (from != 0)
105         # Push the vertex index on the array
106         push!(path, from)
107         # Get the parent vertex index
108         from = vertexTo[from]
109       end
110       # Reverse to obtain natural ordering on the array
111       reverse!(path)
112       # Add the destination vertex index
113       push!(path, to)
114
115       return path
116     end
117
118     # Return distance to destination
119     this.dist = function (to::Int)
120       return distTo[to]
121     end
122
```

```
123     # Initialise fields
124     pq = PriorityQueue()
125     distTo = Array{Float64}(g.V)
126     vertexTo = Array{Int}(g.V)
127
128     # Initialise paths to infinity
129     # Also initialise all vertex path to 0
130     for v = 1 : length(distTo)
131       distTo[v] = Inf
132       vertexTo[v] = 0
133     end
134
135     # Initialise the minimum oriented priority queue with vertex indices
136     for v = 1 : length(g.adj)
137       enqueue!(pq, v, Inf)
138     end
139
140     # Initialise source to 0
141     distTo[s] = 0
142
143     # Set the priority of the source vertex to 0
144     pq[s] = 0
145
146     # Walk through the graph to update distTo thus finding the shortest path
147     # for all other vertices
148     while (!isempty(pq))
149       this.relax(dequeue!(pq))
150     end
151
152     return this
153   end
154 end
155
156 #
157 # Shortest path Floyd-Warshall algorithm
158 #
159 #
160 #
161 type FloydWarshallSP
162   # Variables
163   distTo::Array{Array{Float64}}
164   vertexTo::Array{Array{Edge}}
165
166   # Methods
167   path::Function
168   dist::Function
```

```
169   hasPath::Function
170
171   function FloydWarshallSP(g::EdgeWeightedGraph)
172     this = new()
173
174     # Returns true if there is a path to destination
175     this.hasPath = function (from::Int, to::Int)
176       return distTo[from][to] < Inf
177     end
178
179     # Path as a vertex vector from source to destination
180     # returns a vector of nodes indices to the destination
181     this.path = function (from::Int, to::Int)
182       # Return if there is no path
183       if (!this.hasPath(from, to))
184         return
185       end
186
187       # Build vector going back from destination to source
188       path = Int[]
189       # Get the parent vector closest to destination
190       parent = vertexTo[from][to]
191       while (parent != 0)
192         # Add the vertex to the vector
193         push!(path, parent)
194         # Fetch parent vertex
195         parent = vertexTo[from][parent]
196       end
197       # Reverse the vector order to get natural ordering
198       reverse!(path)
199       # Add destination
200       push!(path, to)
201
202       return path
203     end
204
205     # Return distance to destination
206     this.dist = function (from::Int, to::Int)
207       return distTo[from][to]
208     end
209
210     # Initialise fields
211     distTo = Array{Array{Float64}}(g.V)
212     vertexTo = Array{Array{Int}}(g.V)
213
214     # Initialise paths to infinity
```

```
    # Also initialise all vertex path to 0
    for v = 1 : length(distTo)
      distTo[v] = Array{Float64}(g.V)
      vertexTo[v] = Array{Int}(g.V)
      for w = 1 : length(distTo[v])
        distTo[v][w] = Inf
        vertexTo[v][w] = 0
      end
    end

    # Initialise distances and paths
    for v = 1 : g.V
      for w = 1 : length(g.adj[v])
        e = g.adj[v][w]
        distTo[e.from][e.to] = e.weight
        vertexTo[e.from][e.to] = e.from
      end
      # Handle self loops
      if (distTo[v][v] >= 0.0)
        distTo[v][v] = 0.0
        vertexTo[v][v] = 0
      end
    end

    # Main loop going through the matrix
    for i = 1 : g.V
      for v = 1 : g.V
        # Self loop, don't go
        if (vertexTo[v][i] == 0)
          continue
        end
        # For each neighbour of the vertex
        for w = 1 : g.V
          # alt is the weight path to the neighbour vertex added to the weight
          # from source to that vertex
          alt = distTo[v][i] + distTo[w][i]
          # Update the weights and parent vertex if alt is smaller, meaning it
          # will cost less to pass through that edge
          if (distTo[v][w] > alt)
            distTo[v][w] = alt
            vertexTo[v][w] = vertexTo[i][w]
          end
        end
      end
    end

```

```
261     return this
262   end
263 end
264
265 #
266 # Load graph from a gml file
267 #
268 #
269 #
270 function loadgraph(file::String)
271   # Open the file on disk
272   f = open(file)
273   # Load the contents into a string
274   lines = readlines(f)
275
276   # Load edges in an array
277   edges = Edge[]
278   for l in 1 : length(lines)
279     # Only deal with edge tags
280     if (contains(lines[l], "edge"))
281       edgeLine = l + 2 # Skip edge and [
282       source = ""
283       target = ""
284       weight = ""
285       # Read the edge block until ]
286       while (!contains(lines[edgeLine], "]"))
287         curLine = split(lines[edgeLine])
288         # Avoid incorrect gml
289         if (length(curLine) > 2)
290           println("Unsupported graph")
291           return
292         end
293         # Get the tag values we need
294         if (curLine[1] == "source")
295           source = curLine[2]
296         elseif (curLine[1] == "target")
297           target = curLine[2]
298         elseif (curLine[1] == "value")
299           weight = curLine[2]
300         end # we don't support other tags
301         edgeLine += 1
302       end
303       # If we don't have a source, target or weight for an edge we don't support
304       if ((length(source) == 0) ||
305         (length(target) == 0) ||
306         (length(weight) == 0))
```

```
307           println("Error: Unsupported graph")
308           return
309         else
310           # Add a new edge on the array
311           push!(edges, Edge(parse(Int, source),
312             parse(Int, target), parse(Float64, weight)))
313         end
314       end
315     end
316
317     # Load edges in a map with the source as key to ensure unicity to build
318     # the adjacency matrix
319     adj = Dict()
320     for e in edges
321       # Initialise the edge array
322       if (!haskey(adj, e.from))
323         adj[e.from] = Edge[]
324       end
325       if (!haskey(adj, e.to))
326         adj[e.to] = Edge[]
327       end
328       # Add the edges for the source and target vertex
329       push!(adj[e.from], e)
330       push!(adj[e.to], Edge(e.to, e.from, e.weight))
331     end
332
333     # Convert the dict to the adjacency 2D array
334     fadj = Array{Edge}[]
335     # Create a sorted index vector
336     adjIndices = collect(keys(adj))
337     sort!(adjIndices)
338     for v in adjIndices
339       push!(fadj, adj[v])
340     end
341
342     # Close the file
343     close(f)
344
345     # Return the graph
346     return EdgeWeightedGraph(length(adjIndices), length(edges), fadj)
347   end
348
349   function cout(g::EdgeWeightedGraph, i::Int, j::Int)
350     d = DijkstraSP(g, i)
351     fw = FloydWarshallSP(g)
352
```

```
353   println("Dijkstra")
354   println(d.dist(j))
355   println(d.pathTo(j))
356   println("Floyd-Warshall")
357   println(fw.dist(i, j))
358   println(fw.path(i, j))
359 end
360 
361 #
362 # Main
363 #
364 #
365 #
366 # Load graph from a file where vertices are indexed from 1 to N
367 g = loadgraph("/Users/youri/Downloads/graph.gml")
368 # Or from an adjacency matrix, adj being Array{Array{Edge}}
369 # g = EdgeWeightedGraph(V, E, adj)
370 
371 # Get the shortest path from 1 to 5 on g
372 cout(g, 1, 5)
```